

PROGRAMMING ASSIGNMENT

In this programming assignment you are asked to write parallel solutions to the 4 problems listed below using OpenMp, Pthreads, and Colorama. You do not need to parallelize these codes using Intel Threading Building Blocks, although you are more than welcome, and I certainly will take that into account in your final grade.

Undergraduate students only need to parallelize THREE of these codes using the two approaches (OpenMp, Pthreads or Colorama) of their choice. Notice that the Pthreads and the Colorama version will only differ when critical regions (locks) are necessary. You need to write first a sequential version (sometimes it is provided).

This is an individual work. Codes need to compile and execute properly. Also need to provide input data to tests your codes.

Hand In: Send me a tar file with all the files. The Programming Assignment will be due November 16th.

1. EASY PROBLEM

Parallelization of the following C code:

```
for (i=1; i<n; i++) {  
    s(i) = s(i) + t(i);  
    x(i) = s(i) + w(i);  
    M = M + x(i);  
}
```

Is there anything you can do when parallelizing this code to avoid false sharing?

2. GAME OF LIFE

(From the Wikipedia ...)

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton.

The "game" is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. A variant exists where two players compete. The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- 1) Any live cell with fewer than two live neighbors dies, as if by loneliness.
- 2) Any live cell with more than three live neighbors dies, as if by overcrowding.
- 3) Any live cell with two or three live neighbors lives, unchanged, to the next generation.
- 4) Any dead cell with exactly three live neighbors comes to life.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

Parallelization

To partition this problem we can assume that a thread owns a region consisting of a column panel or a square of a checker board, and each thread computes the values of the next generation for its own region. The parallelization of this problem is very simple if we can have two arrays: one to contain a generation and the other to receive the values (dead or live) of the next generation. These two arrays exchange their roles every generation. In this case, we only need to have a barrier between generations, but critical sections are not needed.

3. SPARSE MATRIX VECTOR MULTIPLICATION

Sparse matrices are those where the number of nonzero elements in the matrix is much less than the number of elements in the matrix. For example, if the matrix has order 10,000 but, on average only 10 nonzeros in each row, then we'll use 100,000,000 doubles or 800 megabytes to store the matrix in the usual format. However, of the 100,000,000 doubles only 100,000 are nonzero. That is, we'll really only be using one one-thousandth of the storage allocated — i.e., 800 kilobytes of storage. There are many solutions to the problem of wasted storage for sparse matrices. In many cases sparse matrices have some special structure, that makes it quite simple to store them efficiently. For example, tridiagonal matrices and block tridiagonal matrices arise naturally in the solution of certain types of PDE's, and these types of matrices can be stored with little information beyond a listing of the nonzero elements. On the other hand, in many other applications (e.g., circuit simulation) the sparse matrices have no special structure. These are the types of matrices we'll be interested in for the programming assignment.

Compressed Sparse Column Format

An obvious data structure for the representation of unstructured sparse matrices is to store in consecutive memory locations each nonzero value along with its row and column indices. While easy to understand, this approach will store 2 integers for each floating point value, and hence duplicate the amount of storage needed if the elements of the matrix are doubles and triplicate the amount of storage needed if they are (single precision) floats.

A more economical alternative is called compressed column row format or CSC format. The idea is to use three arrays:

1. An array of floating point numbers for the nonzeros.
2. An array of integers: the row numbers of the corresponding entry in the array of floating point values.
3. Another array of integers, which stores the subscripts (in the array of floating point numbers) of the first entry in each column.

For example, suppose the matrix is

11	0	0	14	0	16
0	22	0	0	25	26
0	0	33	34	0	36
41	0	43	44	0	46

If the three arrays are called vals, cols, and rows, respectively. Then their contents will be:

```
vals: 11 41 22 33 43 14 34 44 25 16 26 36 46
row:  0  3  1  2  3  0  2  3  1  0  1  2  3
col:  0  2  3  5  8  9 13
```

Thus, the contents of the i -th column are stored in the `vals` array in locations: `col[i], cols[i]+1, cols[i]+2, ... cols[i+1]-1`

Matrix-Vector Multiplication

You are asked to parallelize the sparse-matrix vector multiplication when the matrix is stored using the CSC format and the vectors are stored in the usual (dense) format. So suppose A is stored in CSC format using the arrays `vals`, `cols` and `rows`, as above. A sequential algorithm to multiply matrix A by vector x and store the results in y is shown next:

```
int i, j, k;
```

Initialize vector y to zero.

```
for (i = 0; i < n; i++) {
    for (k = col[i]; k < col[i+1]; k++) {
        j = row[k];
        y[j] += vals[k]*x[i];
    }
}
```

Parallelization

How can we parallelize this algorithm?

Since the matrix is stored by columns, we can assign one or more columns to each thread. However, with this partition several threads can write to the same location of vector y , so writes to y should be protected by a critical section. Another possible solution is to declare a private copy of y in each processor, and then have each processor compute its partial sum. After the loop is done, there is a parallel reduction phase where the contributions of each array are summed to the final array.

Notice that an additional problem of this application is load balancing, since the number of nonzeros in each column can change dramatically, and it could happen that one process/thread is given much more work than the other. A possible solution to this problem that will work well for shared memory is to keep an array of binary markers, one for each column. The two possible values for each marker can be interpreted as “taken” or “not taken”. To start each thread takes a predetermined column or collection of columns. Each of these columns are marked as taken. When a thread runs out of work, it looks for a column (or columns) marked “not taken”.

4. SHORTEST PATH DIJSTRA'S ALGORITHM

(From the Wikipedia ...)

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the single-source shortest path problem for a directed graph with

non negative edge weights.

For example, if the vertices (nodes) of the graph represent cities and edge weights represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between two cities.

The input of the algorithm consists of a weighted directed graph G and a source vertex s in G . We will denote V the set of all vertices in the graph G . Each edge of the graph is an ordered pair of vertices (u,v) representing a connection from vertex u to vertex v . The set of all edges is denoted E . Weights of edges are given by a weight function $w: E \rightarrow (0, \infty)$; therefore $w(u,v)$ is the cost of moving directly from vertex u to vertex v . The cost of an edge can be thought of as (a generalization of) the distance between those two vertices. The cost of a path between two vertices is the sum of costs of the edges in that path. For a given pair of vertices s and t in V , the algorithm finds the path from s to t with lowest cost (i.e. the shortest path). It can also be used for finding costs of shortest paths from a single vertex s to all other vertices in the graph.

Description of the algorithm

The algorithm works by keeping, for each vertex v , the cost $d[v]$ of the shortest path found so far between s and v . Initially, this value is 0 for the source vertex s ($d[s]=0$), and infinity for all other vertices, representing the fact that we do not know any path leading to those vertices ($d[v]=\infty$ for every v in V , except s). When the algorithm finishes, $d[v]$ will be the cost of the shortest path from s to v — or infinity, if no such path exists.

For each vertex, v , the algorithm maintains two sets of vertices, S and Q . Set S contains all vertices for which we know that the value $d[v]$ is already the cost of the shortest path and set Q contains all other vertices. Set S is initially empty, and in each step one vertex is moved from Q to S . This vertex is chosen as the vertex with lowest value of $d[u]$. When a vertex u is moved to S , the algorithm relaxes every outgoing edge (u,v) . That is, for each neighbor v of u , the algorithm checks to see if it can improve on the shortest known path to v by first following the shortest path from the source to u , and then traversing the edge (u,v) . If this new path is better, the algorithm updates $d[v]$ with the new smaller value.

The notion of "relaxation" comes from an analogy between the estimated length of the shortest path and the length of a helical tension spring, which is not designed for compression. Initially, the cost of the shortest path is an overestimate, likened to a stretched out spring. As shorter paths are found, the estimated cost is lowered, and the spring is relaxed. Eventually, the shortest path, if one exists, is found and the spring has been relaxed to its resting length.

C code

```
#define MAX_NODES 1024          /* maximum number of nodes */
#define INFINITY 1000000000    /* a number larger than every maximum path */
int n,dist[MAX_NODES][MAX_NODES]; /*dist[i][j] is the distance from i to j */
void shortest_path(int s,int t,int path[ ]){
    struct state {              /*the path being worked on */
        int predecessor ;      /*previous node */
        int length              /*length from source to this node*/
        enum {permanent, tentative} label /*label state*/
    } state[MAX_NODES];
    int l, k, min;
    struct state *p;

    for (p=&state[0];p < &state[n];p++){ /*initialize state*/
        p->predecessor=-1
        p->length=INFINITY
        p->label=tentative;
    }

    state[t].length=0; state[t].label=permanent ;
    k=t ;                          /*k is the initial working node */
    do{                              /* is the better path from k? */
        for l=0; l < n; l++)          /*this graph has n nodes */
            if (dist[k][l] !=0 && state[l].label==tentative){
                if (state[k].length+dist[k][l] < state[l].length){
                    state[l].predecessor=k;
                    state[l].length=state[k].length + dist[k][l]
                }
            }
        /* Find the tentatively labeled node with the smallest label. */
        k=0;min=INFINITY;
        for (l=0;l < n;l++){
            if(state[l].label==tentative && state[l].length < min){
                min=state[l].length;
                k=l;
            }
        }
        state[k].label=permanent
    }while (k!=s);
    /*Copy the path into output array*/
    l=0;k=0
    Do{path[l++]=k;k=state[k].predecessor;} while (k > =0);
}
```